NumPy Arrays

Main Points

1 Numeric Python (*NumPy*) Introduction

A previous lecture/document used `for` loops or list comprehensions to calculate sequences of numeric values. A more efficient approach is to use *NumPy* `http://www.numpy.org` (short for *Numeric Python*) to perform the same task. The *NumPy* website states that "*NumPy is the fundamental package for scientific computing with Python.*" It does more than just provide for easy creation of and calculations with numeric arrays, it also provides tools for linear algebra, numeric calculus, random numbers, and much more. This document will focus primarily on creating numeric one and two-dimensional `numpy.ndarray` objects.

2 Creation of `numpy.ndarray` Objects

NumPy arrays (the `numpy.ndarray` data type) are powerful objects for working with sequences of numeric values.

- Great for large (even very large) sequences
- Unless a list is quite small, *NumPy* arrays are much more efficient and faster than lists
- From this point on, *NumPy* arrays will be refered to simply as arrays
- Must first import the *NumPy* module to create and use arrays
  - The *NumPy* module is not included with the standard *Python* installation
  - Must download and install it for use with *Python*
    - The *Anaconda* distribution of *Python* includes *NumPy* plus many other scientific modules
    - *NumPy* can be installed alongside a standard *Python* installation using `pip`

- A very popular means of importing *NumPy* is `import numpy as np` – must precede *NumPy* commands with `np.` after importing

- Can adjust the standard display options for values in arrays
  - Use `np.set_printotions()`
  - Default precision is 8 digits, i.e. `precision=8`
  - By default, `suppress=False` where scientific notation will be used if the array includes any very small numbers

Importing *NumPy* and changing display options

```
>>> import numpy as np

>>> # precision=5 will display 5 digits of precision in arrays
>>> # suppress=True will display small values as standard floats
>>> # adding threshold=10 will summarize arrays longer than 10 values
>>> np.set_printoptions(precision=5, suppress=True)
```

2.1 Converting Lists, Tuples, or Ranges to Arrays

Existing *Python* lists, tuples, and ranges can be converted to arrays using the `np.array()` function.

Convert a list to an array

```
>>> x1 = [1.5, 2.7, 3.9]

>>> # convert 'x1'
>>> x1 = np.array(x1)
>>> print(x1)
[1.5 2.7 3.9]

>>> # display 'x1' (implicit printing)
>>> x1
array([1.5, 2.7, 3.9])
```

Convert a range to an array

```
>>> x2 = range(1, 8, 2)

>>> # convert 'x2'
>>> x2 = np.array(x2)

>>> print(x2)
[1 3 5 7]
```

Check the object types of arrays

```
>>> type(x1)
<class 'numpy.ndarray'>

>>> type(x2)
<class 'numpy.ndarray'>
```

2.2 Arrays From Scratch Using `np.array()`

Arrays can also be created from scratch using the `np.array()` function. The argument needs to be a list or tuple containing numeric values.

- All items in a *NumPy* array need to be of the same type
- Being all the same object type helps to make arrays as efficient as they are
- Acceptable to use variables and math operations when creating array items
- If any item in an array is a float, all items in the finished array will also be converted to floats
- Adding a decimal after one value in an array of integers will convert all values to floats
- Cannot replace a single value in an array of integers with a float; the value will be converted to an integer instead, leading to unexpected results

Building arrays from scratch

```
>>> # array of integers
>>> np.array([2, 4, 6, 8, 10])
array([ 2,  4,  6,  8, 10])

>>> # adding one decimal point forces all values to be floats
>>> np.array([2., 4, 6, 8, 10])
array([ 2.,  4.,  6.,  8., 10.])
```

Changing the suppress display option

```
>>> # if one value needs exponential notation, they all show that way
>>> # unless np.set_printoptions(suppress=True)
>>> np.set_printoptions(suppress=False)
>>> np.array([0.0000345, 1500000, 125])
array([3.45e-05, 1.50e+06, 1.25e+02])

>>> np.set_printoptions(suppress=True)
>>> np.array([0.0000345, 1500000, 125])
array([      0.00003, 1500000.     ,     125.     ])
```

Including math when building arrays

```
>>> np.set_printoptions(precision=5, suppress=True)
>>> np.array([1.5, 2/3, 0.0125, 10, 4e-4])
array([ 1.5    ,  0.66667,  0.0125 , 10.     ,  0.0004 ])

>>> cd, ee, h = 6, 3, 4
>>> x = np.array([ee, cd*h, np.cos(np.pi/3), h**2, np.sqrt(h*h/cd), 14])
>>> print(x)
[ 3.      24.       0.5     16.       1.63299 14.     ]
```

2.3 Setting the Type for Array Values

- *NumPy* allows the number type to be specified within the array definition
- Add `float` or `int` as an argument after the list or tuple in `np.array()`
- For example, using `np.array([1, 2, 3, 4], float)` will convert all of the values to floats
- There are other specific precision integers and floats, but standard integers and floats are fine

Explicitly setting the value type for an array

```
>>> np.array([1, 2, 3, 4], float)
array([1., 2., 3., 4.])

>>> # the following will truncate floats into integers
>>> np.array([0, 0.5, 1, 1.5, 2, 2.5], int)
array([0, 0, 1, 1, 2, 2])
```

3 Creating Arrays of Linearly Spaced Values

3.1 The `np.arange()` Function

The `np.arange()` function operates much like the built-in `range()` function except the result is a *NumPy* array instead of a list-like range object.

- `np.arange()` can be used to create arrays using starting, ending, and step values
  - If the step argument is omitted, then the step size will be 1
  - If only one argument is used the array will start at zero and have a step of 1
  - The constructed array will not include the ending value
- A negative step will allow you to create an array where the values decrease from start to the end
- One important difference between `np.arange()` and `range()` is the ability to use floats
- Use `np.arange()` when a specific step size is known

Creating *NumPy* arrays using `np.arange()`

```
>>> # 1 to 7 (including 7) with a step of 1
>>> np.arange(1, 8)
array([1, 2, 3, 4, 5, 6, 7])

>>> # 10 to 1 (including 1) with a step of -0.5
>>> np.arange(10, 0.5, -0.5)
array([10. ,  9.5,  9. ,  8.5,  8. ,  7.5,  7. ,  6.5,  6. ,  5.5,  5. ,
        4.5,  4. ,  3.5,  3. ,  2.5,  2. ,  1.5,  1. ])

>>> # -5 to 1 with a step of 1/3
>>> np.arange(-5, 1.1, 1/3)
array([-5.     , -4.66667, -4.33333, -4.     , -3.66667, -3.33333,
       -3.     , -2.66667, -2.33333, -2.     , -1.66667, -1.33333,
       -1.     , -0.66667, -0.33333, -0.     ,  0.33333,  0.66667,
        1.     ])
```

3.2 The `np.linspace()` Function

The `np.linspace()` function also creates arrays of evenly spaced values. The biggest difference from `np.arange()` is that `np.linspace()` returns an array with a specified number of values instead of a predetermined step size.

- The arguments passed to the function are `(start, stop, num)`
  - `num` is the total number of values to create
  - If `num` is omitted it will default to 50 values
  - There are optional keyword arguments
    - `endpoint=` defaults to `True`, causing the array to include the `stop` value
    - `dtype=` forces the array to use a specific type
- Use `np.linspace()` when the number of values in an array is known or required

Using `np.linspace()` to create arrays

```
>>> # 10 float values from 1 to 10, inclusive
>>> np.linspace(1, 10, 10)
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])

>>> # 5 values from 30 to 0, inclusive
>>> np.linspace(30, 0, 5)
array([30. , 22.5, 15. ,  7.5,  0. ])

>>> # 20 values from 0 to pi, inclusive
>>> np.linspace(0, np.pi, 20)
```

```
array([0.     , 0.16535, 0.33069, 0.49604, 0.66139, 0.82673, 0.99208,
       1.15743, 1.32278, 1.48812, 1.65347, 1.81882, 1.98416, 2.14951,
       2.31486, 2.4802 , 2.64555, 2.8109 , 2.97625, 3.14159])
```

4 Using *NumPy* Math Functions

When creating and working with *NumPy* arrays the *NumPy* versions of math functions should be used (if available), *not* the `math` module versions.

- The *NumPy* versions can work on all values in an array but `math` cannot
- For example, use `np.pi`, `np.sin()`, `np.exp()`, `np.sqrt()`, etc
- Another document will cover this topic in greater depth

Using *NumPy* math functions on/with arrays

```
>>> array_1 = np.array([np.sqrt(5),
...                     np.sin(np.pi/3),
...                     np.degrees(np.arctan(1.0))])
>>> array_1
array([ 2.23607,  0.86603, 45.     ])

>>> array_2 = np.arange(6)
>>> np.sqrt(array_2)
array([0.     , 1.     , 1.41421, 1.73205, 2.     , 2.23607])
```

5 Two-Dimensional Array Creation

Two-dimensional arrays are simply arrays of arrays.

- Only one `np.array()` function is required
  - Place lists separated by commas within a slist
  - Example: `np.array([[1, 2], [3, 4]])`
  - Each list must be the same size to avoid problems
- It is possible to create a 2D array that has only one row
  - For a one-dimensional array use `np.array([1, 2, 3])`
  - Use an additional set of square brackets to create a 2D array
  - For example: `np.array([[1, 2, 3]])`

The following code block creates and then prints the following 2D arrays.

$$A1 = \begin{bmatrix} 5 & 35 & 43 \\ 4 & 76 & 81 \\ 21 & 32 & 40 \end{bmatrix} \qquad A2 = \begin{bmatrix} 7 & 2 & 76 & 33 & 8 \\ 1 & 98 & 6 & 25 & 6 \\ 5 & 54 & 58 & 9 & 0 \end{bmatrix}$$

Creating 2D arrays

```
>>> A1 = np.array([[5, 35, 43], [4, 76, 81], [21, 32, 40]])

>>> print(A1)
[[ 5 35 43]
 [ 4 76 81]
 [21 32 40]]

>>> A2 = np.array([[7, 2, 76, 33, 8],
...                [1, 98, 6, 25, 6],
...                [5, 54, 58, 9, 0]])

>>> print(A2)
[[ 7  2 76 33  8]
 [ 1 98  6 25  6]
 [ 5 54 58  9  0]]
```

Both `np.arange()` and/or `np.linspace()` functions can be used when creating any of the rows in a 2D array. A four row 2D array has been created in the following code block as described below using `np.arange()` and `np.linspace()` where appropriate.

1. 1 to 11 (inclusive) in steps of 2
2. A step size of 5 from 0 to 25 (inclusive)
3. 6 linearly spaced values from 10 to 60 (inclusive)
4. The list `[67, 2, 43, 68, 4, 13]`

Creating a 2D array with `np.arange()` and `np.linspace()`

```
>>> a = np.array([np.arange(1, 12, 2),
...               np.arange(0, 26, 5),
...               np.linspace(10, 60, 6),
...               [67, 2, 43, 68, 4, 13]])

>>> print(a)
[[ 1.  3.  5.  7.  9. 11.]
 [ 0.  5. 10. 15. 20. 25.]
 [10. 20. 30. 40. 50. 60.]
 [67.  2. 43. 68.  4. 13.]]
```

6 Some Array Attributes for Size and Shape

NumPy arrays also have associated methods much like *Python* lists have. They also have attributes, which look like methods without parentheses. However, they return information about an object instead of operating like a function on/with the object. Following are some *NumPy* array attributes associated with array size and shape.

- `ndarray.ndim` is used to find the number of dimensions of an array
  - Given `x = np.array([[1, 2],[3, 4]])`
  - `x.ndim` will return a value of 2 since `x` is a 2D array
- `ndarray.size` returns the total number of items in the array
  - `x.size` will return a value of 4 for the previous array
- `ndarray.shape` returns the size of the array in each direction
  - For 2D arrays `ndarray.shape` returns the tuple `(rows, columns)`
  - `x.shape` results in `(2, 2)` since there are 2 rows and 2 columns in the array

Checking size, shape, and dimensions of a 1D array

```
>>> a = np.array([1, 2, 3])
>>> print(a)
[1 2 3]

>>> a.size
3
>>> a.shape
(3,)
>>> a.ndim
1
```

Checking size, shape, and dimensions of a one row 2D array

```
>>> b = np.array([[1, 2, 3]])
>>> print(b)
[[1 2 3]]

>>> b.size
3
>>> b.shape
(1, 3)
>>> b.ndim
2
```

Checking size, shape, and dimensions of a two row 2D array

```
>>> c = np.array([[1, 2, 3], [8, 9, 10]])
>>> print(c)
[[ 1  2  3]
 [ 8  9 10]]

>>> c.size
6
>>> c.shape
(2, 3)
>>> c.ndim
2
```

7 Special Array Creation Functions

The following *NumPy* functions can be used to create special types of arrays.

- `np.zeros(n)` will create a 1D array filled with `n` 0s
- `np.zeros((r, c))` will create an `r` $\times$ `c` 2D array filled with `n` 0s
- `np.ones()` function works like `np.zeros()` except that it creates an array filled with 1s
- `np.eye(n)` function creates an `n` $\times$ `n` 2D identity array
  - Filled with zeros except ones on the diagonal from the top-left to the bottom-right
  - Must be square – same number of rows and columns
  - `np.eye(3)` will create a 3$\times$3 identity array
- `np.diag()` is like an identity array except diagonal is filled with the values from the argument...
  - List
  - Tuple
  - Array

The following arrays are created, assigned to variables, and displayed in the code block below using one of the special array creation functions.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Diagonal array from the following vector... $\begin{bmatrix} 12 & 13 & 15 & 18 \end{bmatrix}$

Creating arrays with special array creation functions

```
>>> np.zeros((2, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])

>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

>>> np.ones((5, 3))
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])

>>> a = [12, 13, 15, 18]   # use a list, tuple, or array
>>> np.diag(a)             # could create list in the function
array([[12,  0,  0,  0],
       [ 0, 13,  0,  0],
       [ 0,  0, 15,  0],
       [ 0,  0,  0, 18]])
```

8 Math with *NumPy* Arrays and Scalars

Unlike built-in *Python* lists, *NumPy* arrays can be used directly to perform math operations. Specifically, you can add, subtract, multiply, or divide scalar values by/with arrays. Other operations are also possible, but these will be looked at in more detail in another document.

Array-scalar math

```
>>> f = np.ones(5)
>>> f*5
array([5., 5., 5., 5., 5.])
>>> f/5
array([0.2, 0.2, 0.2, 0.2, 0.2])

>>> g = np.arange(1, 6)
>>> g + 100
array([101, 102, 103, 104, 105])
```

9 *NumPy* Array Functions and Methods

9.1 Size and Shape Reporting Functions

- The `len()` function will return...
  - The number of objects in a one-dimensional array
  - The number of rows (lists) in a two-dimensional array
- `np.size()` returns the total number of objects in one or two-dimensional arrays
  - This function does the same thing as the `.size` attribute from earlier
  - Given `z = np.array([[1, 4], [5, 8]])`, `np.size(z)` will return 4
- `np.shape()` returns a tuple with the number of rows and columns for an array
  - This function does the same thing as the `.shape` attribute used previously
  - If 1D, it will return a tuple with one item that represents the length of the array
- `np.ndim()` does the same thing as the `.ndim` attribute; returning the number of dimensions

Array size and shape functions for a 1D array

```
>>> AA = np.array([5, 9, 2, 4])

>>> len(AA)
4
>>> np.size(AA)
4
>>> np.shape(AA)
(4,)
>>> np.ndim(AA)
1
```

Array size and shape functions for a 2D array

```
>>> BB = np.ones((4, 3))

>>> len(BB)
4
>>> np.size(BB)
12
>>> np.shape(BB)
(4, 3)
>>> np.ndim(BB)
2
```

9.2 Changing an Array Shape

- `np.reshape(array, (new_row, new_col))`
  - Makes a copy of the array and changes the copy's shape to a new row and column size
  - The arguments are...
    - The array to be reshaped
    - A tuple containing the new shape
  - The new array shape must have the same total number of elements as original array
  - The shape is changed by...
    - Placing all of the existing rows into one single row in order starting with row zero
    - This 1D array is reshaped by creating new rows
    - Each new row will have the number of items as set by the new column argument
  - The `ndarray.reshape(new_row, new_col)` method does the same thing as this function
  - Both `np.reshape()` (and `ndarray.reshape()`) can auto-calculate rows or columns
    - If `-1` is used for either the `new_row` or `new_col` arguments
    - The correct value for that direction will be based on the value of the other argument
    - Total number of items are divided by the specified argument to determine the number of items for the `-1` direction
    - If 12 objects, `new_row` is `-1`, and `new_col` is 3, then the result will have 4 rows
- The `ndarray.resize()` method does the same thing as `ndarray.reshape` except...
  - Does not make a copy, it changes the original array
  - Does not allow `-1` to be used as an argument
- `np.transpose()` function can change shape of two-dimensional arrays
  - Transposing an array changes the shape by trading the rows and columns
    - Existing first row will become the new first column
    - Existing first column will become the new first row
  - The `ndarray.T` (transpose) attribute does the same thing
  - Transposing creates a copy instead of changing the original array

Using the `np.reshape()` function

```
>>> B = np.array([[5, 1, 6],
...               [8, 0, 2]])

>>> np.reshape(B, (3, 2))
array([[5, 1],
       [6, 8],
       [0, 2]])

>>> B       # original does not change
array([[5, 1, 6],
       [8, 0, 2]])
```

Using the `np.reshape()` function with auto-calculation

```
>>> np.reshape(B, (-1, 2))
array([[5, 1],
       [6, 8],
       [0, 2]])

>>> B      # still does not change
array([[5, 1, 6],
       [8, 0, 2]])
```

Using the `ndarray.reshape()` method

```
>>> B.reshape(3, 2)
array([[5, 1],
       [6, 8],
       [0, 2]])

>>> B      # still not changing
array([[5, 1, 6],
       [8, 0, 2]])
```

Using the `ndarray.resize()` method

```
>>> B.resize(3, 2)

>>> B      # this changes the original array
array([[5, 1],
       [6, 8],
       [0, 2]])
```

Using `np.transpose()` on an array

```
>>> np.transpose(B)
array([[5, 6, 0],
       [1, 8, 2]])
>>> B      # original does not change
array([[5, 1],
       [6, 8],
       [0, 2]])
```

Using `ndarray.T` on an array

```
>>> B.T
array([[5, 6, 0],
       [1, 8, 2]])

>>> B      # original still does not change
array([[5, 1],
       [6, 8],
       [0, 2]])
```

Using the `ndarray.reshape()` method on a one-dimensional array is a good way to create a two-dimensional array.

Reshaping a 1D array into a 2D array

```
>>> A = np.array([5, 1, 6, 8, 0, 2])
>>> A
array([5, 1, 6, 8, 0, 2])

>>> A.reshape(3, 2)
array([[5, 1],
       [6, 8],
       [0, 2]])

>>> A      # notice that the original is unchanged
array([5, 1, 6, 8, 0, 2])
```

Resizing a 1D array into a 2D array

```
>>> A.resize(3, 2)

>>> A      # notice that the original does change this time
array([[5, 1],
       [6, 8],
       [0, 2]])
```

10 Array Addressing (Indexing)

Addressing individual items in *NumPy* arrays works the same way as for *Python* lists.

- One-dimensional arrays
  - Place a set of square brackets with the desired index position after an array or array name
  - For example, `arr1[0]`

- Two-dimensional arrays
  - The index needs to include two values; row and column
  - For example, `arr2[0, 2]` will access the item located in row 0 at the index 2 position
  - Can also use `arr2[0][2]` to get the value at the 2nd index position of the the 0th row
- Negative indices work can be used as well

Indexing a 1D array

```
>>> vct = np.array([35, 46, 78, 23, 5, 14, 81, 3, 55])
>>> vct
array([35, 46, 78, 23,  5, 14, 81,  3, 55])

>>> vct[4]
5

>>> vct[6]
81

>>> vct[3] + vct[-2]    # 23 + 3 = 26
26
```

Indexing a 2D array

```
>>> MAT = np.array([[3, 11, 6, 5], [4, 7, 10, 2], [13, 9, 0, 8]])
>>> MAT
array([[ 3, 11,  6,  5],
       [ 4,  7, 10,  2],
       [13,  9,  0,  8]])

>>> # first value from the last row (the number 13)
>>> MAT[-1, 0]
13

>>> # last value in the 2nd row minus 2nd value in the first row (2 - 11)
>>> MAT[1, -1] - MAT[0, 1]
-9
```

11 Array Slicing

- Slicing 1D *NumPy* arrays is done the same way as for lists and strings
- Slicing 2D arrays is similar to 1D arrays
  - The colon is still used to separate starting, ending, and step indices for slicing
  - Since there are 2 dimensions, slice must to be done for the rows and columns
  - For example, `arr2[row_start:row_end, col_start:col_end]`
  - The slice `arr2[0:2, 0:3]` would return the first two rows of the first three columns
  - All objects in a row or column are included if there is only a colon in the row or column slice
  - For example, `arr2[:, 1:3]` returns the items from all rows with column indices of 1 and 2

Slicing a 1D array

```
>>> v1 = np.array([4, 15, 8, 12, 34, 2, 50, 23, 11])

>>> # slice v1 to create the array with [8, 12, 34, 2, 50]
>>> u1 = v1[2:7]
>>> u1
array([ 8, 12, 34,  2, 50])
```

Slicing all rows or columns from a 2D array

```
>>> MA = np.array([np.arange(1, 12, 2),
...                np.arange(2,13,2),
...                np.arange(3,19,3),
...                np.arange(4,25,4),
...                np.arange(5, 31, 5)])
>>> MA
array([[ 1,  3,  5,  7,  9, 11],
       [ 2,  4,  6,  8, 10, 12],
       [ 3,  6,  9, 12, 15, 18],
       [ 4,  8, 12, 16, 20, 24],
       [ 5, 10, 15, 20, 25, 30]])

>>> # all rows of the third column
>>> MA[:, 2]
array([ 5,  6,  9, 12, 15])

>>> # all columns of the first row
>>> MA[2]      # same as MA[2, :]
array([ 3,  6,  9, 12, 15, 18])
```

Slicing some rows and/or columns from a 2D array

```
>>> # row indices 1 thru 3 of all columns
>>> MA[1:4]      # same as MA[1:4, :]
array([[ 2,  4,  6,  8, 10, 12],
       [ 3,  6,  9, 12, 15, 18],
       [ 4,  8, 12, 16, 20, 24]])

>>> # column indices 2 thru 4 of all rows
>>> MA[:, 2:5]
array([[ 5,  7,  9],
       [ 6,  8, 10],
       [ 9, 12, 15],
       [12, 16, 20],
       [15, 20, 25]])

>>> # intersection of row indices 0 thru 2 and column indices 1 thru 3
>>> MA[:2, 1:4]
array([[3, 5, 7],
       [4, 6, 8]])

>>> # every other row of every other column
>>> MA[::2, ::2]
array([[ 1,  5,  9],
       [ 3,  9, 15],
       [ 5, 15, 25]])
```

12 Replacing Values of Objects in Arrays

NumPy arrays are mutable, just like lists. Array indexing and slicing can be used to replace (change) individual values or groups of values in arrays. If the array has an integer data type, attempting to replace a value with a float will result in an integer instead.

Replacing a single array value

```
>>> AX = np.arange(0,25,3)
>>> AX
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])

>>> AX[1] = 42
>>> AX
array([ 0, 42,  6,  9, 12, 15, 18, 21, 24])
```

Replacing multiple array values

```
>>> AX[3:7] = np.zeros(4)     # replacing 4 values with zeros
>>> AX
array([ 0, 42,  6,  0,  0,  0,  0, 21, 24])
```

13 Adding Values to Arrays

- Use `np.append()` to extend the length of a 1D array by adding items to the end
  - The arguments are the original array followed by the appended array or range
  - Returns a copy of the original array
  - Must assign the appended copy to the original variable name in order to "change" the original
  - Adding and removing values to/from *NumPy* arrays is not very computationally efficient
  - It is generally better to redefine an array with the correct values than to change an array

Appending to a 1D array

```
>>> DF = np.arange(1, 5)

>>> # extend DF with 10 to 35 (inclusive) with steps of 5
>>> DF = np.append(DF, np.arange(10, 36, 5))
>>> DF
array([ 1,  2,  3,  4, 10, 15, 20, 25, 30, 35])
```

- Appending values to 2D arrays takes more work; `np.append()` accepts an optional `axis` argument
  - It is `None` by default, but can be set to `0` or `1` when working with 2D arrays
  - When `axis=0` is used
    - A row is added
    - The list that is appended needs to be placed in double square brackets
  - When `axis=1` is used, a column is added
    - The items being added to each row need their own square brackets

Appending to a 2D array

```
>>> E = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

>>> F = np.append(E,[[9, 10, 11, 12]], axis=0)
>>> F
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> G = np.append(F,[[44],[45],[46]], axis=1)
```

```
>>> G
array([[ 1,  2,  3,  4, 44],
       [ 5,  6,  7,  8, 45],
       [ 9, 10, 11, 12, 46]])
```

- The `np.insert()` function can be used to add elements at locations other than the end
  - This function also returns a copy of the original array
  - The arguments are...
    - The array
    - The index position before which insertion will happen
    - The value or values to insert
  - The index position can be given as a list of index positions
    - Doing so requires a list with the same number of values as there are being inserted
    - Each will be inserted before the index positions relative to the original array

Inserting values into an array

```
>>> r = np.ones(6)

>>> r = np.insert(r, 2, 3)
>>> r
array([1., 1., 3., 1., 1., 1., 1.])

>>> np.insert(r, 0, [2, 3, 4])
array([2., 3., 4., 1., 1., 3., 1., 1., 1., 1.])

>>> s = np.arange(6)
>>> s
array([0, 1, 2, 3, 4, 5])

>>> np.insert(s, 1 , np.arange(5,8))
array([0, 5, 6, 7, 1, 2, 3, 4, 5])

>>> np.insert(np.arange(9, 0, -1), [1, -1], [1, 9])
array([9, 1, 8, 7, 6, 5, 4, 3, 2, 9, 1])

>>> np.insert(np.array([5]), 0, np.zeros(6))
array([0, 0, 0, 0, 0, 0, 5])

>>> np.append(np.zeros(6), 5)
array([0., 0., 0., 0., 0., 0., 5.])
```

- The `np.hstack()` function can join together 2 or more 1D arrays
  - This function accepts a tuple of arrays, lists, and/or individual values as an argument
  - It connects them together horizontally in an end-to-end fashion
- Another clever way to perform tasks similar to `np.hstack()` is to use `np.r_[]`
  - Can be used to join arrays, lists, tuples, ranges, and/or values by extending the row length
  - Square brackets are used instead of standard parentheses
  - There is also a `np.c_[]` command that concatenates by extending the column length

Using `np.hstack()` and `np.r_[]` to build arrays

```
>>> np.hstack(([99, 1000], np.arange(0, 5, 1.5), 999))
array([  99. , 1000. ,    0. ,    1.5,    3. ,    4.5,  999. ])

>>> np.r_[np.zeros(3), 5, (99, 100), np.arange(5)]
array([  0.,   0.,   0.,   5.,  99., 100.,   0.,   1.,   2.,   3.,   4.])
```